# Package 'pipeflow'

June 15, 2026

**Title** Fast Interactive Data Analysis Pipelines

**Version** 0.3.0

**Maintainer** Roman Pahl <roman.pahl@gmail.com>

**Description** A lightweight and intuitive framework for building interactive data analysis pipelines. You add R functions one by one, and 'pipeflow' wires them into a pipeline that stays consistent as you go. Modify, remove, or insert steps at any stage, manage all parameters in one place, fast execution (C++-powered DAG) for interactive use and Shiny backends.

**License** MIT + file LICENSE

**URL** https://rpahl.github.io/pipeflow/, https://github.com/rpahl/pipeflow

**BugReports** https://github.com/rpahl/pipeflow/issues

**Depends** R (>= 4.2.0)

**Imports** data.table, jsonlite, lgr, methods, R6, Rcpp (>= 1.1.1), stats, utils

**Suggests** ggplot2, gridExtra, knitr, mockery, rmarkdown, targets, testthat, visNetwork

**LinkingTo** Rcpp

**VignetteBuilder** knitr

**Config/roxygen2/version** 8.0.0

**Config/testthat/edition** 3

**Config/testthat/parallel** true

**Encoding** UTF-8

**Language** en-US

**NeedsCompilation** yes

**Author** Roman Pahl [aut, cre]

**Repository** CRAN

**Date/Publication** 2026-06-15 07:50:07 UTC

# Contents



---

`length.pipeflow_pip` *Length of a pipeflow pipeline or view*

---

## Description

Length of a pipeflow pipeline or view

## Usage

```
## S3 method for class 'pipeflow_pip'
length(x)

## S3 method for class 'pipeflow_view'
length(x)
```

## Arguments

| | |
|---|---|
| `x` | A pipeflow pipeline or view |

## Value

Number of steps as an integer.

## Examples

```
p <- pip_new() |>
  pip_add("s1", \(x = 1) x) |>
  pip_add("s2", \(x = ~s1) x + 1) |>
  pip_add("s3", \(x = ~s2) x * 2)
length(p) # 3 — total steps in the pipeline

# A view reports only the number of selected (visible) steps
v <- pip_view(p, i = c("s2", "s3"))
length(v) # 2
```

---

# pip_add *Add a step*

---

## Description

Adds a named step to the pipeline. Each step is a function whose parameters either hold constant defaults or reference the output of a prior step using formula notation (`~step_name`). Dependencies are validated when the step is added.

## Usage

```
pip_add(x, step, fun, tags = character(0), after = length(x), exec = "auto")
```

## Arguments

| | |
|---|---|
| `x` | A pipeflow pipeline object. |
| `step` | Unique step name. |
| `fun` | Function to execute for the step. Each function parameter must have a default value. Default values that are simple constants are resolved immediately. Default values that are formulas like `~other_step` are treated as dependencies to those steps and resolved to the respective output values at runtime once the step is executed. |
| `tags` | Optional character vector of tags belonging to the step. Can also be adjusted later using `[pip_tag()]`. |
| `after` | Optional position after which the new step should be inserted (defaults to last position). Can be a step name or an integer index. If set to 0, the new step will be inserted at the beginning of the pipeline. |
| `exec` | Execution mode for this step. One of "auto", "split", "reduce" or "plain". Using execution mode `exec = split`, the output of the step is marked as partitioned output. In this mode, any step that depends on the split step (directly or indirectly) will have its output automatically mapped partition-wise during step |

execution. The `reduce` mode expects partitioned input and passes it through without mapping, while `plain` mode only accepts non-partitioned input and always intends to execute a single call. In summary:

- auto: map if partitioned input appears, otherwise single call
- split: single call, then mark output as partitioned
- reduce: single call, but only valid with partitioned input
- plain: single call, only valid with non-partitioned input

## Details

If `after` was specified, the new step will be inserted after the given step or position. Be aware that in contrast to adding a step at the end, inserting a step in the middle is a rather expensive operation as it requires re-wiring parts of the internal pipeline structure, especially if the new step is inserted at an early position.

## Value

The updated pipeline, invisibly.

## Examples

```
# --- Tags, and view filtering ---
p <- pip_new("analysis") |>
  pip_add("load", \(n = 5) seq_len(n), tags = c("io", "raw")) |>
  pip_add("clean", \(x = ~load) x * 2, tags = c("io", "process")) |>
  pip_add("fit", \(x = ~clean) sum(x), tags = c("model", "core", "daily")) |>
  pip_add("report", \(x = ~fit) paste("result:", x), tags = "report")

pip_run(p)
p

# Filter by tag using pip_view — keeps steps with any matching tag
pip_view(p, tags = "daily")
pip_view(p, tags = "core")
pip_view(p, tags = c("raw", "report"))

# --- Split / reduce execution modes ---
q <- pip_new("split-demo") |>
  pip_add("data", \(x = iris) x) |>
  pip_add("split", \(x = ~data) split(x, x$Species),
    exec = "split"
  ) |>
  pip_add("stats", \(x = ~split) summary(x)) |>
  pip_add("combine", \(x = ~stats) do.call(rbind, x),
    exec = "reduce"
  )

pip_run(q)
q[["stats", "out"]]   # partitioned list — one summary per species
q[["combine", "out"]] # combined table
```

## pip_add_from *Copy a step from another pipeline*

### Description

Copies one step from pipeline `y` into pipeline `x`, preserving its function, parameters, tags, and dependency links.

### Usage

```
pip_add_from(x, y, step)
```

### Arguments

| | |
|---|---|
| `x` | Target pipeflow pipeline object. |
| `y` | Source pipeflow pipeline object. |
| `step` | Step name to copy from `y`. |

### Value

The updated target pipeline, invisibly.

### Examples

```
# Build a source pipeline with reusable steps
src <- pip_new("source") |>
  pip_add("load", \(n = 3) seq_len(n)) |>
  pip_add("square", \(x = ~load) x^2)

# Copy steps into a new pipeline one at a time.
# The dependency of "square" on "load" is re-established automatically.
dst <- pip_new("target")
pip_add_from(dst, src, "load")
pip_add_from(dst, src, "square")
pip_run(dst)
pip_collect_out(dst)
```

## pip_bind *Bind pipelines*

### Description

Bind two pipelines together by concatenating their steps. If both pipelines have steps with the same name, the step names of the second pipeline will be automatically adapted to avoid name clashes.

## Usage

```
pip_bind(x, y)
```

## Arguments

| | |
|---|---|
| x | A pipeflow pipeline object. |
| y | A pipeflow pipeline object. |

## Value

A new pipeflow pipeline object representing the bound pipelines.

## Examples

```
a <- pip_new("a") |>
  pip_add("prep", \(x = 1) x * 2) |>
  pip_add("fit", \(x = ~prep) x + 10)

# "prep" exists in both pipelines; the one from b gets a numeric suffix
b <- pip_new("b") |> pip_add("prep", \(x = 5) x * 3)

ab <- pip_bind(a, b)
ab[["step"]] # "prep", "fit", "prep2" (step name conflict auto-resolved)
ab
```

---

# pip_clone *Clone a pipeline*

---

## Description

Creates an independent copy of the pipeline. Changes to the cloned pipeline do not affect the original pipeline, and vice versa.

## Usage

```
pip_clone(x, name = NULL)
```

## Arguments

| | |
|---|---|
| x | A pipeflow pipeline object. |
| name | Optional name for the cloned pipeline. If NULL, the original name is used. |

## Value

A cloned pipeflow pipeline object.

### Examples

```
p <- pip_new("original") |>
  pip_add("s1", \(x = 1) x) |>
  pip_add("s2", \(x = ~s1) x + 1)

# Clone produces a fully independent copy
cp <- pip_clone(p, name = "copy")
pip_add(cp, "s3", \(x = ~s2) x * 10)

# As a result, the clone has the new step ...
cp

# ... while the original is left unchanged
p
```

---

## pip_collect_out *Collect step outputs*

---

### Description

Returns the outputs of all pipeline steps as a flat named list keyed by step name. Use pip_view() to narrow the selection before collecting, and compose calls if grouped output is needed.

### Usage

```
pip_collect_out(x)
```

### Arguments

| | |
|---|---|
| x | A pipeflow pip or view. |

### Value

A named list of outputs, one element per step.

### Examples

```
p <- pip_new() |>
  pip_add("load", \(x = 1) x, tags = "io") |>
  pip_add("clean", \(x = ~load) x + 1, tags = "io") |>
  pip_add("model", \(x = ~clean) x * 2, tags = "model")
pip_run(p)

# Flat named list with one entry per step
pip_collect_out(p)

# Combine with pip_view to collect output for specific tags
grouped <- list(
  io = pip_view(p, tags = "io") |> pip_collect_out(),
```

```
  model = pip_view(p, tags = "model") |> pip_collect_out()
)
grouped
```

---

`pip_get_graph` *Build pipeline graph data*

---

## Description

Builds graph data (nodes and edges) describing the pipeline's step structure, suitable for visualisation with `visNetwork::visNetwork()`.

## Usage

```
pip_get_graph(x, include_upstream = FALSE)
```

## Arguments

| | |
|---|---|
| `x` | A pipeflow pip or view. |
| `include_upstream` | Logical. Only relevant for views. If `TRUE`, add all upstream dependencies of selected steps. |

## Details

Node shapes reflect execution mode:

- `auto`/`plain`: `hexagon`
- `reduce`: `dot`
- `split`: `star`

## Value

A named list with two `data.frames`: `nodes` and `edges`.

## Examples

```
p <- pip_new()
pip_add(p, "load", \(x = 1) x, tags = "io")
pip_add(p, "clean", \(x = ~load) x + 1, tags = "io")
pip_add(p, "fit", \(x = ~clean) x * 2, tags = "model")

graph <- pip_get_graph(p)
graph$nodes # data.frame: id, label, shape, color
graph$edges # data.frame: from, to, arrows

# For a view, include_upstream = TRUE adds upstream deps to the graph
v <- pip_view(p, i = "fit")
```

```
pip_get_graph(v, include_upstream = TRUE)

if (require("visNetwork", quietly = TRUE)) {
  do.call(what = visNetwork::visNetwork, args = graph)
}
```

---

`pip_get_params` *Get independent parameters*

---

## Description

Returns the current default values of all tunable (non-dependency) parameters across the pipeline. These are the parameters that can be updated via `pip_set_params()`. Parameters wired to another step's output via `~step_name` are excluded.

## Usage

```
pip_get_params(x)
```

## Arguments

| | |
|---|---|
| `x` | A pipeflow pip or view |

## Value

Named list of tunable parameter values. If the same parameter name appears in multiple steps, the first occurrence in pipeline order is returned.

## Examples

```
p <- pip_new() |>
  pip_add("load", \(n = 100, seed = 42) seq_len(n)) |>
  pip_add("model", \(x = ~load, lambda = 0.1) x * lambda)

# ~load is a dependency — only non-dependency params are returned
pip_get_params(p) # list(n = 100, seed = 42, lambda = 0.1)

# Useful as a guide for pip_set_params()
pip_set_params(p, params = list(n = 20, lambda = 0.5))
pip_run(p) |> pip_collect_out()
```

## pip_has_step *Check whether a step exists*

### Description

Check whether a step exists

### Usage

```
pip_has_step(x, step)
```

### Arguments

| | |
|---|---|
| x | A pipeflow pip |
| step | A step name |

### Value

Logical indicating if the step exists

### Examples

```
p <- pip_new() |>
  pip_add("load", \(x = 1) x) |>
  pip_add("fit", \(x = ~load) x + 1)

pip_has_step(p, "load") # TRUE
pip_has_step(p, "fit") # TRUE
pip_has_step(p, "predict") # FALSE — step not yet added
```

## pip_lock *Lock selected steps against updates*

### Description

Locks all selected steps in the pipeline unless p is a view, in which case only steps covered by the view are locked.

### Usage

```
pip_lock(p)
```

### Arguments

| | |
|---|---|
| p | A pipeflow pip or view. |

## Value

The updated pipeline or view, invisibly.

## Examples

```
p <- pip_new() |>
  pip_add("load", \(x = 10) x) |>
  pip_add("fit", \(x = ~load) x * 2)
pip_run(p, lgr = NULL)

# Lock only "load" via a view so it won't be re-executed or overwritten
pip_lock(pip_view(p, i = "load"))
p[["pipeline"]][["locked"]] # TRUE, FALSE

# Locked steps are silently skipped during pip_run()
pip_run(p, lgr = NULL, force = TRUE)
p[["pipeline"]][["out"]][[1]] # still 10 — locked, not re-executed

pip_unlock(p)
p[["pipeline"]][["locked"]] # FALSE, FALSE
```

---

# pip_new *Create a pipeline*

---

## Description

Creates a new, empty pipeline. Add steps with `pip_add()` and execute them with `pip_run()`.

## Usage

```
pip_new(name = "pipe")
```

## Arguments

| | |
|---|---|
| name | Single name used for printing and for derived view names. |

## Value

A pipeflow pipeline object.

## Examples

```
# Create a named pipeline
p <- pip_new("my_analysis")
p[["name"]] # "my_analysis"

# Build a simple pipeline and run it
pip_add(p, "load", \(n = 5) seq_len(n))
pip_add(p, "double", \(x = ~load) x * 2) # x depends on load's output
```

```
p
pip_run(p)
p[["out"]] # list of outputs, one per step
```

---

`pip_remove` *Remove a step*

---

## Description

If other steps depend on the step to be removed, an error is given and the removal is blocked, unless `recursive` was set to `TRUE`. In recursive mode, the selected step and all downstream dependent steps are removed together.

## Usage

```
pip_remove(x, step, recursive = FALSE)
```

## Arguments

| | |
|---|---|
| `x` | A pipeflow pip |
| `step` | `string` the name of the step to be removed. |
| `recursive` | `logical` if `TRUE` the step is removed together with all its downstream dependencies. |

## Value

The updated pipeline, invisibly.

## Examples

```
p <- pip_new() |>
  pip_add("load", \(x = 1) x) |>
  pip_add("transform", \(x = ~load) x * 2) |>
  pip_add("model", \(x = ~transform) x + 10)

# Removing a leaf step (nothing depends on it) works directly
pip_remove(p, "model")
p                        # "load", "transform"

# Trying to remove a step that others depend on raises an error:
# pip_remove(p, "load")  # Error!

# recursive = TRUE removes the step and all its downstream dependents
pip_remove(p, "load", recursive = TRUE)
p                        # pipeline is now empty
```

## pip_rename *Rename a step*

### Description

Renames the selected step and updates dependency references in downstream steps.

### Usage

```
pip_rename(x, from, to)
```

### Arguments

| | |
|---|---|
| x | A pipeflow pip |
| from | Existing step name |
| to | New step name |

### Value

The updated pipeline, invisibly.

### Examples

```
p <- pip_new() |>
  pip_add("s1", \(x = 1) x) |>
  pip_add("s2", \(x = ~s1) x + 1)           # "s2" depends on "s1"

# Downstream dependency references are updated automatically
pip_rename(p, from = "s1", to = "load_data")
p

#' # Trying to rename to an existing step name raises an error:
try(pip_rename(p, "load_data", to = "s2"))  # step 's2' already exists!
```

## pip_replace *Replace a step*

### Description

Replaces a step's function while keeping it in the same position in the pipeline. Downstream steps are automatically marked as outdated and will re-run on the next pip_run().

### Usage

```
pip_replace(x, step, fun, tags = character(0))
```

### Arguments

| | |
|---|---|
| x | A pipeflow pipeline object. |
| step | Step name. |
| fun | Function to execute for the step. |
| tags | Optional character vector of tags belonging to the step. Can also be adjusted later using [pip_tag()]. |

### Value

The updated pipeline, invisibly.

### Examples

```
p <- pip_new() |>
    pip_add("load", \(n = 5) seq_len(n)) |>
    pip_add("double", \(x = ~load) x * 2)
pip_run(p)
p

# Replace "load" — downstream steps are automatically marked "outdated"
pip_replace(p, "load", \(n = 3) seq_len(n))
p

# Re-run to bring everything up to date
pip_run(p)
p
```

---

## pip_run *Run a pipeline*

---

### Description

Executes all pending steps in order. Steps already in state "done" are skipped unless force = TRUE.

### Usage

```
pip_run(
  x,
  lgr = pipeflow_lgr,
  force = FALSE,
  progress = NULL,
  recursive = FALSE
)
```

## Arguments

| | |
|---|---|
| x | A pipeflow pip or view |
| lgr | A logging function of the form function(level, msg, ...). To suppress logging, you can set lgr = NULL. |
| force | Logical indicating if all steps should be forced to run, regardless of whether they are outdated or not. |
| progress | Optional callback of the form function(value, detail) called before each step. |
| recursive | If TRUE and a step returns a pipeline object, the current run is aborted and continues from the returned pipeline. Useful for dynamic or self-modifying pipelines. |

## Details

When x is a view, requested rows are run together with required upstream dependencies.

## Value

The updated pipeline or view, invisibly.

## See Also

vignette("v06-self-modify-pipeline", package = "pipeflow") for an advanced example of recursive/dynamic pipelines.

## Examples

```
p <- pip_new() |>
  pip_add("load", \(n = 3) seq_len(n)) |>
  pip_add("square", \(x = ~load) x^2) |>
  pip_add("total", \(x = ~square) sum(x))

pip_run(p)
p

# Already-done steps are skipped on a second run
pip_run(p) # all steps skipped

# lgr = NULL suppresses log output
pip_run(p, lgr = NULL)

# force = TRUE re-executes every step regardless of state
pip_run(p, force = TRUE)

# Run only a subset of steps via a view;
# upstream dependencies are automatically included
v <- pip_view(p, i = "total")
pip_run(v)
```

`pip_set_params` *Set independent parameters*

## Description

Updates the default values of tunable parameters across the pipeline. Affected steps and their downstream dependents are automatically marked as outdated.

## Usage

```
pip_set_params(p, params = list())
```

## Arguments

| | |
|---|---|
| `p` | A pipeflow pip or view |
| `params` | Named list of parameters to set. |

## Details

Parameters of locked steps are never changed and their state remains unchanged.

## Value

The updated pipeline or view, invisibly.

## Examples

```
p <- pip_new() |>
  pip_add("load", \(n = 10) seq_len(n)) |>
  pip_add("scale", \(x = ~load, factor = 0.5) x * factor)

# See all adjustable parameters before running
pip_get_params(p) # list(n = 10, factor = 0.5)

# Updating params marks affected steps (and their dependents) outdated
pip_set_params(p, params = list(n = 5, factor = 2.0))
p

pip_run(p)
p
```

## pip_tag *Add tags to selected steps*

### Description

Adds tags to existing tags for all steps in the pipeline unless p is a view, in which case tags are only added for steps covered by the view. Locked steps are skipped and not updated.

### Usage

```
pip_tag(p, tags = character())
```

### Arguments

| | |
|---|---|
| p | A pipeflow pip or view. |
| tags | Character vector of tags to add for each selected step. |

### Value

The updated pipeline or view, invisibly.

### Examples

```
p <- pip_new() |>
  pip_add("load", \(x = 1) x) |>
  pip_add("fit", \(x = ~load) x + 1)

# Tag every step in the pipeline at once
pip_tag(p, tags = c("daily", "core"))
p[["pipeline"]][["tags"]] # both steps have c("daily", "core")

# Add an extra tag to only one step via a view
v <- pip_view(p, i = "fit")
pip_tag(v, tags = "model")
p[["pipeline"]][["tags"]] # "fit" also has "model"
```

## pip_unlock *Unlock selected steps*

### Description

Unlocks all selected steps in the pipeline unless p is a view, in which case only steps covered by the view are unlocked.

### Usage

```
pip_unlock(p)
```

### Arguments

| | |
|---|---|
| p | A pipeflow pip or view. |

### Value

The updated pipeline or view, invisibly.

### Examples

```
p <- pip_new() |>
  pip_add("load", \(x = 1) x) |>
  pip_add("fit", \(x = ~load) x * 2)

# Lock all steps, then unlock to restore normal execution
pip_lock(p)
p[["pipeline"]][["locked"]] # TRUE, TRUE

pip_unlock(p)
p[["pipeline"]][["locked"]] # FALSE, FALSE
```

---

## pip_untag *Remove tags from selected steps*

---

### Description

Removes tags from existing tags for all steps in the pipeline unless p is a view, in which case tags are only removed for steps covered by the view. Locked steps are skipped and not updated.

### Usage

```
pip_untag(p, tags = character())
```

### Arguments

| | |
|---|---|
| p | A pipeflow pip or view. |
| tags | Character vector of tags to remove for each selected step. |

### Value

The updated pipeline or view, invisibly.

## Examples

```
p <- pip_new() |>
  pip_add("load", \(x = 1) x, tags = c("daily", "core")) |>
  pip_add("fit", \(x = ~load) x + 1, tags = c("daily", "model"))

# Remove "daily" from all steps
pip_untag(p, tags = "daily")
# "load" retains "core"; "fit" retains "model"
p[["pipeline"]][["tags"]]
```

---

# pip_view *Create a pipeline view*

---

## Description

Creates a filtered view showing only a selected subset of steps. A view references the underlying pipeline without copying it, so operations like `pip_run()` and `pip_set_params()` applied to a view affect only the selected steps.

## Usage

```
pip_view(
  x,
  i = integer(),
  filter = list(),
  tags = character(),
  fixed = TRUE,
  ...
)
```

## Arguments

| | |
|---|---|
| `x` | A pipeflow pipeline or view. |
| `i` | Optional row indices or step names to keep. |
| `filter` | A named list of filters to apply. Each element can be a character vector specifying the values to keep for the corresponding property or, if `fixed` is FALSE, a regular expression. See examples for usage. |
| `tags` | Tag filter (character). Keeps steps with any matching tag. |
| `fixed` | If TRUE, values in `filter` are treated as fixed strings, otherwise they are treated as regular expressions. |
| `...` | further args passed to `grepl` (only in effect when `fixed` is `FALSE`). |

## Value

A `pipeflow_view` object.

## Examples

```
p <- pip_new()
pip_add(p, "load_raw", \(x = 1) x,
  tags = c("io", "core", "daily")
)
pip_add(p, "fit_model", \(x = 2) x + 1,
  tags = c("model")
)
pip_add(p, "eval_model", \(x = ~fit_model) x,
  tags = c("model", "daily", "report")
)

# Filter by a fixed column value (one or more states)
pip_view(p, filter = list(state = "new"))

# Combine filters: step pattern AND state
pip_view(p, filter = list(step = "model", state = "new"))

# Filter by tag — keeps steps that have *any* of the given tags
pip_view(p, tags = "daily")

# Combine explicit step selection with a filter (intersection)
pip_view(p,
  i      = c("load_raw", "fit_model"),
  filter = list(state = "new")
)

# Select by integer row indices
pip_view(p, i = c(1L, 2L), filter = list(state = "new"))

# Use a regex pattern to match step names
pip_view(p, filter = list(step = "_model$"), fixed = FALSE)

# Views are composable: create a view-of-view for progressive narrowing
v1 <- pip_view(p, tags = "daily")
print(v1) # load_raw, eval_model
v2 <- pip_view(v1, tags = "report")
print(v2) # eval_model only
```

---

# print.pipeflow_pip *Print pipeflow objects*

---

## Description

Print pipeflow objects

## Usage

```
## S3 method for class 'pipeflow_pip'
```

```
print(
  x,
  rows = integer(),
  cols = getOption("pipeflow.print.cols", default = "core"),
  topn = getOption("pipeflow.print.topn", default = 5),
  nrows = getOption("pipeflow.print.nrows", default = 50),
  row.names = getOption("pipeflow.print.rownames", default = TRUE),
  class = getOption("pipeflow.print.class", default = FALSE),
  header = TRUE,
  ...
)

## S3 method for class 'pipeflow_view'
print(x, header = TRUE, ...)
```

## Arguments

| | |
|---|---|
| x | A pipeflow pipeline or view. |
| rows | Row indices to be printed. If empty, all rows are printed. |
| cols | The columns to be printed. Can be either one of `core` or `all` to print the core or all columns, respectively, or an explicit character vector of columns to be printed. |
| topn | The number of rows to be printed from the beginning and end of tables with more than `nrows` rows. |
| nrows | The number of rows printed before truncation is enforced. |
| row.names | If TRUE, row indices will be printed alongside x. |
| class | If TRUE, the resulting output will include above each column its storage class (or a self-evident abbreviation thereof). |
| header | If TRUE, a header with the pipeline name and number of steps will be printed. |
| ... | Other arguments passed to `print.data.table` |

## Value

Invisibly returns `x`.

## Examples

```
p <- pip_new("demo") |>
  pip_add("load", \(n = 5) seq_len(n), tags = c("io", "raw")) |>
  pip_add("square", \(x = ~load) x^2, tags = "compute") |>
  pip_add("total", \(x = ~square) sum(x), tags = "compute")

print(p) # core columns: step, depends, tags, out, state
print(p, cols = "all") # all non-hidden columns
print(p, rows = 2:3) # print only steps 2 and 3
p <- pip_new() |>
  pip_add("s1", \(x = 1) x, tags = "io") |>
  pip_add("s2", \(x = ~s1) x + 1, tags = "model")
```

```
# A view header shows how many steps are selected out of the total
v <- pip_view(p, tags = "model")
print(v) # "<pipeflow_view> pipe view (1 of 2 steps)"
```

---

`set_log_layout` *Set pipeflow log layout*

---

## Description

This function provides an easy way to set the basic log layout of the pipeline logging. For a fine-grained control of the logger, which you can retrieve via `lgr::get_logger("pipeflow")`, see e.g. the logger_config function from the lgr package.

## Usage

```
set_log_layout(layout = c("text", "json"))
```

## Arguments

| | |
|---|---|
| `layout` | Layout name, which at this point can be either 'text' or 'json'. |

## Value

invisibly returns a `Logger` object

## Examples

```
p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(data = ~data, x = 1) x + data)
p$run()

lg <- set_log_layout("json")
print(lg)

p$run()

set_log_layout("text")
p$run()
```

`[.pipeflow_pip` *Extract or subset a pipeline*

## Description

Returns a new pipeline containing selected steps and all required upstream dependencies.

Extracts values from a pipeline using one or two indices. With a single string name, named fields such as `"pipeline"` or `"name"` are returned first; anything else returns the matching step-table column. With two indices (`row`, `column`), a single cell is extracted.

## Usage

```
## S3 method for class 'pipeflow_pip'
x[i, ...]

## S3 method for class 'pipeflow_pip'
x[[i, j, ...]]
```

## Arguments

| | |
|---|---|
| `x` | A pipeflow pipeline object. |
| `i` | integer (row indices) or character vector (step names) of steps to select |
| `...` | not used |
| `j` | column names to select |

## Value

A new pipeflow pipeline object.

Extracted value(s), depending on `i` and `j`.

## Examples

```
p <- pip_new() |>
  pip_add("load", \(n = 5) seq_len(n)) |>
  pip_add("square", \(x = ~load) x^2) |>
  pip_add("total", \(x = ~square) sum(x))

# Select by step name — upstream deps are pulled in automatically.
# Selecting only "total" still includes "load" and "square".
sub <- p["total"]
sub[["pipeline"]][["step"]] # "load", "square", "total"

# Select a subset of steps by name vector
p[c("load", "square")][["pipeline"]][["step"]] # "load", "square"

# Select by integer row index
p[1:2][["pipeline"]][["step"]] # "load", "square"
```

```
p <- pip_new() |>
  pip_add("load", \(x = 1) x) |>
  pip_add("fit", \(x = ~load) x + 1)

# Access internal objects by name
p[["pipeline"]] # the full step table
p[["name"]] # "pipe"

# Shorthand column access (equivalent to p[["pipeline"]][["step"]])
p[["step"]]

# Two-index form: p[[row, column]] extracts a single cell
p[["fit", "depends"]] # "load"
p[[2, "state"]] # state of the second step
```

# Index